LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-671193

Lessons Learned from Implementing OMPD: a Debugging Interface for OpenMP

J. Protze, I. Laguna, D. H. Ahn, J. DelSignore, A. Burton, M. Schulz, M. S. Mueller

May 21, 2015

International Workshop on OpenMP (IWOMP)
Aachen, Germany
October 1, 2015 through October 2, 2015

Disclaimer

Lessons Learned from Implementing OMPD: a Debugging Interface for OpenMP ⋆

Joachim Protze[1,2,3], Ignacio Laguna[3], Dong H. Ahn[3], John DelSignore[4], Ariel Burton[4], Martin Schulz[3], and Matthias S. Müller[1,2]

[1] RWTH Aachen University, D-52056 Aachen, Germany
[2] JARA – High-Performance Computing, D-52062 Aachen, Germany
{protze, mueller}@itc.rwth-aachen.de
[3] Lawrence Livermore National Laboratory, Livermore, CA 94550, USA
{lagunaperalt1, ahn1, schulzm}@llnl.gov
[4] Rogue Wave Software, Bolder, CO 80301, USA
{ariel.burton, john.delsignore}@roguewave.com

Abstract. With complex codes moving to systems of increasing on-node parallelism using OpenMP, debugging these codes is becoming increasingly challenging. While debuggers can significantly aid programmers, existing ones support OpenMP at a low system-thread level, reducing their effectiveness. The previously published draft for a standard OpenMP debugging interface (OMPD) is supposed to enable the debuggers to raise their debugging abstraction to the conceptual levels of OpenMP by mediating the tools and OpenMP runtime library. In this paper, we present our experiences and the issues that we have found on implementing an OMPD library prototype for a commonly used OpenMP runtime and a parallel debugger.

1 Introduction

OpenMP is becoming increasingly popular as a portable programming model for *on-node parallelism* as programmers desire to port their codes to its simple directive-based API. This trend, however, is presenting great challenges to debugging. As OpenMP enables easy mapping of tasks to a wide range of resources—more cores, wider simultaneous multithreading (SMT), single-instruction/multiple-data (SIMD) units and accelerators like GPUs and co-processors—reasoning about the OpenMP program's state when debugging can quickly overwhelm programmers.

A parallel debugger is an effective aid to guide programmers in inspecting the state of parallel programs. Programmers can follow through source lines and easily examine the state of key variables at arbitrary points in execution. While today's debuggers support debugging of OpenMP programs at a low system-thread level, they do not allow debugging at the level that programmers conceive the high-level programming model abstractions. For example, no existing debugger provides support, such as stepping a

⋆ Part of this work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. (LLNL-CONF-671193)

logically-related group of threads together (which requires identifying the teams of threads that are at the same OpenMP parallel nesting level), displaying the conceptual stack trace of a thread (e.g., by splicing the trace of a thread to that of the master thread and to identify and omit the trace belonging to the OpenMP runtime itself), and showing the state in which an OpenMP thread could be in.

To effectively aid programmers, debuggers must raise their debugging abstraction to the conceptual level of OpenMP. Constructing the conceptual state requires, however, that the debuggers are able to extract at runtime relevant information from the OpenMP's runtime system. An existing approach that is used in commercial debuggers, such as TotalView [6] or Allinea DDT [1], is to build the knowledge necessary to interact with each runtime directly into the debugger. While useful, it has led to limited support in terms of use cases and of compiler and runtime implementations of the OpenMP language. A standard interface approach, in which debuggers can extract the relevant state from *any* OpenMP runtime system, can lead to a much better solution.

In this paper, we report on our early analysis and experiences with OMPD, the standard OpenMP debug interface recently proposed by the OpenMP Tools Committee [4]. As much as we desire OMPD to serve as the general interface, we have found that it presents obstacles and challenges as we implement it for the Intel OpenMP Runtime, a popular OpenMP runtime library, as well as for TotalView and GDB, widely used debuggers. Thus, we discuss modifications to the current specification needed to overcome our issues. We hope that our experiences will shed light on the effective OpenMP debugging path to other debugger and OpenMP runtime implementers.

In the rest of the paper, we first summarize prior work, and then describe the OMPD interface and its functional architecture. Next, we illustrate some of the important use cases that OMPD can enable. Then, we discuss the problems that we have encountered with the current specification of OMPD and propose our suggested modifications. Finally, we describe future challenges and conclude.

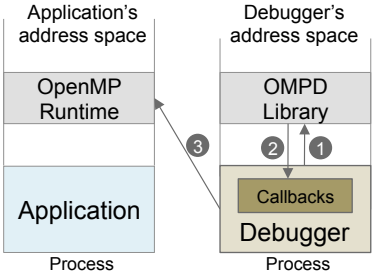

Fig. 1. Overview of the workflow of OMPD: (1) the debugger requests information about OpenMP (e.g., the state of an OpenMP thread, parallel region, task, etc.) via an OMPD API function call; (2) OMPD calls back the debugger to request information of the OpenMP runtime (e.g., the value of a symbol in the runtime); (3) the debugger gets this information from the runtime.

2 Prior Work

Previous work has proposed portable debugging interfaces for parallel runtime libraries, such as for MPI [3] and threads [7]. The key mechanism to providing portability is the encapsulation of the debugging API in a loadable library, which forms the bridge between debuggers and runtime systems. This library is dependent on the internal implementation details of a particular runtime and is loaded by the debugger to request information from that runtime. Upon initialization, the debugger registers hooks with this library to provide the necessary functionality to access the target process (e.g., using the trace interface). When the debugger then calls the debugging API implemented by this library, it uses these hooks to extract information from the target process and then uses its knowledge about internal information to interpret it and return it to the debugger. This decouples debugging functionality implemented by the debugger from implementation dependent runtime information.

Crownie and Gropp used this design to implement an interface that allows a debugger to obtain the information necessary to display the contents of MPI message queues [3]. The same concept has been used in the `libthread_db` library [7], an interface for monitoring and inspecting thread-related aspects of multithreaded programs. Similar thread debugging interfaces have been implemented on other systems such as Tru64, IRIX, AIX and Linux.

Crownie et al. proposed `DMPL` [2], an interface to help a debugger understand the internals of the OpenMP runtime using the aforementioned library-based design. The focus of this interface is to allow a debugger obtain information about *shared* and *private* variables of OpenMP parallel regions.

More recently, the OpenMP Tools Committee proposed `OMPD` [4], a general interface to debug OpenMP programs. `OMPD` extends the functionality proposed in `DMPL` and covers a wider range of debugging use cases—from examining the state of OpenMP threads and tasks, to allowing the debugger to place breakpoints at the beginning and end of parallel regions. We describe these use cases later in this paper.

3 The OpenMP Debugging Interface

Tools targeting OpenMP need access to state information within the OpenMP runtime to improve their ability to deal with OpenMP abstractions and to provide information to users at that level of abstraction. This is true for both performance and debugging tools, although, different requirements apply.

3.1 OMPT: A Runtime Interface for OpenMP Tools

The standard way to provide state information from a runtime is a set of additional API functions exposed by the runtime. The recently proposed OMPT interface [5], which has also been published by the OpenMP ARB as an official white paper, takes this approach and offers both state query functions and callback functionality for relevant events. This can be used by performance tools to examine the state of the runtime, identify parallel regions and tasks, and to assemble call stacks to offer users a view without interwoven runtime stack frames.

3.2 Why Distinguish OMPD from OMPT?

In general, the information offered by OMPT is also required by debuggers. However, debuggers access information in a fundamentally different way: they access and debug a process externally from a different process. This is commonly referred to as "third-party access", which makes a pure runtime library API approach difficult. In particular, calling a function in the target process's runtime has the following problems:

- It may not be possible at all, for example, it is not possible to call a function in a target core file. Some target architectures, such as GPUs, may not permit the debugger to call functions at all.
- It is unreliable since the target process or thread may have corrupted itself to the point where calling a function causes a crash (e.g., a `SIGSEGV`). Also, it assumes that the function being called in the target is asynchronously reentrant.
- It may change the process and thread state and may have unintended side effects.
- It is relatively expensive and may scale poorly, as each call requires many low-level operations to read/write target memory and registers, continue execution, handle breakpoint traps, and cleanup the process.

OMPD therefore intends to expose the same information as available from OMPT or from the OpenMP APIs directly, but without the need to run in the process of the application and the runtime. As shown in Figure 1, the OMPD runtime is loaded by the debugger and resides in the debugger process. The OMPD library supports the debugger in getting the right information out of the runtime library.

Note that an OpenMP library does not necessarily need to implement OMPT to provide OMPD. We therefore propose to use a distinct namespace for each of the two interfaces and do not reuse type names from the OMPT interface for OMPD (even if they offer similar or identical semantics). For example, the header file `ompd.h` should not depend on `ompt.h`. Nevertheless, OpenMP runtime implementers may choose to implement a common internal state tracking for OMPT and OMPD.

3.3 The OMPD Architecture

As illustrated in Figure 1, the OMPD library is loaded by the debugger. Whenever the debugger calls an OMPD API function, the library needs to get information from the OpenMP runtime library (e.g., reading values from the runtime library's memory). To maintain a clear separation of concerns, this functionality is not provided (i.e., reimplemented) by the OMPD library; rather the debugger exposes functions for the OMPD library to access the address space of the target process. As the debugger loads the OMPD library, the debugger registers these functions as callback functions with the OMPD library, so the library can access the functions. Using these debugger callback functions, the OMPD library accesses the memory space of the target process, which contains the needed OpenMP state information. It then uses its constructed knowledge of the runtime system and OpenMP objects to return the requested information to the debugger. Section 5 focuses on the set of callback functions defined for OMPD.

Special issues emerge when the debugger process runs on an architecture different from where the target application with the OpenMP runtime library is run. Examples for

```
1 {
2 // code before parallel region
3 #pragma omp parallel
4   {
5     // parallel region code
6   }
7 // code after parallel region
8 }
```

Listing 1.1. Parallel region is the basic OpenMP constuct

this situation include debugging on a system like IBM Blue Gene/Q or Cray systems, where the compute nodes have a different architecture from the front-end nodes, or in a hybrid system combining CPUs with accelerators such as GPUs. We will address some of these issues in Sections 6 and 7.

4 Use Cases of OMPD

OpenMP-aware debuggers must enable a programmer to debug the program at the level of OpenMP programming abstractions. Tools must be capable of making the relevant program state visible without revealing unnecessary implementation details. OMPD must be designed and implemented to empower tools to serve this purpose. In this section, we present some of the capabilities that an OpenMP-aware debugger can provide and how OMPD can be used to help the debugger provide these capabilities. They are the representative use cases that cannot easily be supported without the help of a standard runtime debug interface.

4.1 OpenMP-Aware Stack Trace

One of the most important debugging views of a thread is its stack trace. For OpenMP programs, however, the raw stack trace of a thread has proven to be inadequate because it often contains too much detail on the underlying OpenMP runtime implementation while not fully capturing its high-level semantics.

Let's take an example of a basic construct in OpenMP: a parallel region. Listing 1.1 shows a minimal code example. For each thread that arrives at the begin of the parallel region, a team of threads are created, which will execute in parallel the block following the `omp parallel` pragma. Listing 1.2 shows how this `omp parallel` pragma can be translated. This source-to-source translation represents how a compiler could realize the occurrence of this high-level OpenMP pragma into a low-level mechanism.

The function `omprt_run_parallel` in this example is implemented within the OpenMP runtime library and is responsible for creating the team of threads and for making these threads execute the function being passed as a function argument. This is a commonly used technique although each runtime may use a distinct function name for `omprt_run_parallel`: `main._omp_fn` is used in GNU OpenMP while `.omp_microtask.` is used in the Intel OpenMP Runtime.

For an execution of the code that has been translated according to this scheme, at least two distinct stack traces can result. The master thread, which is the thread that

```
1  void parallel_region_block()
2  {
3    // parallel region code
4  }
5  [...]
6  {
7  // code before parallel region
8  omprt_run_parallel(parallel_region_block);
9  // code after parallel region
10 }
```

Listing 1.2. Simplified source-to-source translation of pragma omp parallel

```
in parallel_region_block () from file:3
in omprt_internal () from libopenmp
in omprt_run_parallel () from libopenmp
in block () from file:8
```

Listing 1.3. Stack trace of the master thread pausing at a breakpoint on line 3 of Listing1.2

created the team, will have a stack trace shown in Listing 1.3. A stack trace for any other team member (or slave) thread is shown in Listing 1.4. Note that this is a simplified example for illustration purpose only. Depending on the runtime implementation, the stack trace might appear much more obfuscated.

There are two main problems with the representation shown in this example. First, the slave thread stack trace lacks the history about the parallel-region context. Listing 1.4 provides no clue that `parallel_region_block` originated from the parallel region or called from within `block`. Even if the raw stack trace of a system-level thread includes a thread creation history, often this would not help either. Most runtime implementations manage thread pools and reuse threads across teams so the history information could be mixed up.

More importantly, from a programmer's perspective, this raw representation of a stack trace is inadequate: most programmers do not want to see all the runtime internal indirections in the stack trace. Instead, an abstraction at the conceptual level of the programmer's model, as shown in Listing 1.5, is more insightful. To provide such a high-level representation, however, the debugger needs to fetch the following information from a runtime debug interface: (1) the hierarchy of parallel contexts; (2) the entry and exit points of the runtime to unroll the parallel-region creation of the master thread and to remove runtime library functions from the stack trace.

For this purpose, OMPD provides the function `ompd_get_enclosing_parallel_handle`, which allows a debugger to unroll the hierarchy of parallel contexts. We will describe this

```
in parallel_region_block () from file:3
in omprt_internal () from libopenmp
in start_thread () from libpthread
in clone () from libc
```

Listing 1.4. Stack trace for a team member pausing at a breakpoint on line 3 of Listing1.2

```
in #omp parallel from file:5 @ T3
in block () from file:3 @ T1
```

Listing 1.5. Stack trace as it shoud be provided for team member thread 3

OMPD function in details in Section 6, and a modification to the current specification, which we need to improve this workflow.

4.2 Stepping In and Out of a Parallel Region

Another common use case when working with a debugger is stepping through the execution, entering, and leaving functions. For the example in Listing 1.1 the user would expect to reach line 5 when stepping in, and line 7 when stepping out of the parallel region. This is the behavior when the code is compiled without OpenMP enabled. However, with OpenMP and without special handling by the debugger, a single step would end up in the OpenMP runtime library. Instead, the debugger must again hide the implementation details of the runtime library, moving forward to the reentry point in the application.

To enable this, OMPD must supply entry point information at the right place. When entering the parallel region, the region is not created yet, so the information is unavailable. The expectation is that, at some point between entering the runtime and leaving the runtime, the information about the entry point to the application is available. The debugger then needs to stop the execution of the target only when this information becomes available, extract the information and continue to the entry point. Thus, OMPD must provide breakpoint information to the debugger so that it can be notified via a breakpoint event only when the information is available. We will discuss in Section 6 how OMPD should provide the breakpoint information.

5 OMPD callback interface

Here we describe the callback functions that a debugger needs to provide to the OMPD library to enable the library to gather information from the application. We carefully reduce the set of callback functions to a minimum and discuss where we see issues with the set provided by the current OMPD document [4].

5.1 Functions for Operating System Interaction

A lesson learned from prior debugging libraries is that a library that is loaded by a debugger should not rely on system memory management, but instead use debugger-provided memory management. Using the primitive memory management callback functions `ompd_alloc_memory` and `ompd_free_memory`, the library gives the debugger control over its memory management. This allows the debugger to use its own custom implementation of memory management (e.g., malloc/free vs. new/delete).

Similarly it is best practice to use the debugger's output routines for output that is produced by the library. This way the debugger can redirect the output in its usual way,

for example to `stderr` or to a log file. The callback function `ompd_print_string` provides a simple interface to print strings using the debugger's output stream.

The current OMPD proposal defines a function to resolve an error code to a string. As the error codes are specified in the interface, the string should be constant and well defined. Consequently, there is no reason why the error string should be provided by the debugger. Thus we propose to remove this callback function.

5.2 Resolving Structures for Target Architecture

In general, we cannot assume that the OMPD library and the OpenMP runtime operate on the same architecture. On the other hand, we do not want to see multiple OMPD libraries that are specialized built for each target architecture. For these reasons, the library needs a way to get the sizes of target types at runtime. The debugger knows about the target architecture, so we assume a debugger should be able to provide size information for primitive types that are defined in the C standard with an architecture and compiler dependent size. The OMPD callback function `ompd_sizeof_prim_ttype` is defined to return a vector of sizes for the types `char`, `short`, `int`, `long`, `long long`, and the pointer type `void *`.

The draft of the OMPD interface suggests functions to resolve application specific structs and functions to get sizes and offsets for structure elements. This approach is in general not applicable as most runtime libraries are delivered in a stripped format with removed type information. On the other hand, the information about structure sizes and member offsets cannot be calculated within the OMPD library when the application is executed on a different architecture. Further, an OMPD library should be able to handle OpenMP runtime libraries built for different architectures, so the OMPD library needs to get the information about structure sizes and offsets from the targeted runtime library.

The pthread debugging interface [7] does not use callbacks for resolving types either. The approach for the pthread library is to include all necessary sizes and offsets in the runtime library—they can be calculated during initialization of the library and can be fetched by reading the value of integer global symbols. For the pthread library, this is implemented using preprocessor macros to transparently provide and access the sizes when new symbols are added to structures. An OMPD library implementation might use a similar macro approach or just put all the needed offsets in the code. While our proposed change to OMPD does not specify *how* structure offsets and sizes are calculated by OMPD, it omits the callbacks for structure type and member lookups.

5.3 Access Application Memory

The API function `ompd_tsymbol_addr_lookup` is used to identify the base address for any basic symbol in the address space of the application respectively the OpenMP runtime library. We will discuss implications of the access to thread local or accelerator address space in Section 5.4.

Based on the address of a symbol and offsets for elements, the OMPD library will use the memory access functions `ompd_read_tmemory` and `ompd_write_tmemory` to read and write values in target memory. We propose to use an additional argument to specify the primitive type for the access and replace the

size argument with a count argument that specifies the number of array items. With this information, the debugger might perform endianness conversion for a memory access.

The current OMPD callback interface suggests a function to convert the endianness of memory, which is read from the target memory before. This function misses an argument to specify the primitive type for the conversion or misses the argument to express the count of values. A reason for having a dedicated function for read and type conversion is that reading from the target memory can have a quite high latency. Reads of multiple values from a struct in the target memory would have the latency for every read. The debugger might cache the memory page and reduce the latency. On the other hand, we expect just the read of single values or vectors of values since this is the amount of information returned in the API functions.

We propose to specify the primitive type instead of a size, to give the debugger the possibility to distinguish pointer from integer values. The return type for pointer reads should be `ompd_taddr_t`.

5.4 Debugger's Context Argument

Most API and callback functions include a context pointer. For the debugger the context pointer identifies on which target process, thread, or address space the callback function is supposed to operate. The debugger provides the context pointer when calling an API function. The OMPD library must pass the context pointer back to the debugger as an argument to most of the callback functions.

In general, the OMPD library should not assume that a context pointer is valid after the API call returned. The state of the target application might change or the debugger might use the handle in another way. The key question is: where does an OpenMP implementation store the values to answer the OMPD API function call? Thus, what context is needed to answer the question?

For OMPT, the answer is simple: the API function is called in a thread context, thus the function must be answered with information available in this thread's context. For SMP systems, an OMPD library should be able to answer API function calls with knowledge from the corresponding thread. Thus, the debugger must provide the right thread context with each API function call.

The current version of the tools interface did not consider the target construct and the use of OpenMP with accelerators. The information might be stored on the thread that initiated the `target` region or in the thread on the accelerator. The debugger cannot know where the information is stored. Nevertheless, the debugger needs to provide the context pointer to interact with the right address space. OMPD needs a callback function to request the right context, so that for example, it can navigate from an accelerator context to a process context in search for accelerator-thread information.

6 OMPD API Function Specifications

In this section, we describe high-level problems we expect with the OMPD API specification as proposed in the first technical report on OMPT[4].

6.1 Providing Information on Compatible Runtime Library

The technical report does not specify a way to tell the debugger how to find a compatible OMPD library for a runtime library. It suggests that the OpenMP runtime might provide a list of filename strings that identify the locations of all the compatible OMPD library implementations. This approach will fail when we think of heterogeneous systems with running the application on one architecture and operation system, and debugging on another platform. For example, we cannot expect that the runtime library on the compute nodes would carry the OMPD path information on the login nodes. Our proposal is to give a unique name to each OMPD library in terms of the version and optional architecture information corresponding to the runtime library. The debugger would then attempt to find this OMPD library in the systems library path.

6.2 API Specification for Breakpoints

As described in Section 4.2, OMPD needs to provide breakpoint information for all the cases where control gets transferred to the OpenMP runtime, especially for entering and leaving parallel regions and tasks.

The current OMPD specification has a structure containing four pointers to code locations where the debugger might set breakpoints to get notified of the entering and leaving event of parallel regions and tasks. However, the four locations might be insufficient to cover general cases. A runtime library might have multiple implementations of handling parallel regions for various corner cases; or an OpenMP implementation does not outline the parallel region as shown in our Listing 1.2 but inlines the runtime code. In the latter case, a breakpoint for every parallel region is necessary.

Another fundamental issue is that this approach is not extensible. For example, there might be a need for a new breakpoint for the `target` construct. Changing this `struct`, however, will break compatibility between interface versions.

From the debugging perspective, it's more scalable to have a constant symbol for all parallel processes and threads than collecting addresses from all processes to place breakpoints.

For all these reasons, we propose to specify the names of dummy breakpoint functions, which need to be called by the OpenMP runtime to trigger the events. The dummy function is an empty function, but the runtime library needs to make sure that the compiler does not optimize out the function call. The debugger then sets the breakpoints to these functions within the runtime whenever needed:

```
1 void ompd_break_pre_parallel(){}
2 void ompd_break_post_parallel(){}
3 void ompd_break_pre_task(){}
4 void ompd_break_post_task(){}
```

Extending this list would not break compatibility with the previous interface.

6.3 Missing Function to Identify Master

When creating a stack trace as described in 4.1, the debugger needs to resolve the parent thread for a parallel region. The OpenMP standard has the function `ompd_get_anchestor_thread_num` to get the parent thread for the parallel region.

```
1 EXTERN ompd_rc_t ompd_get_anchestor_thread(
2   ompd_context_t *context, /* IN: debugger handle for the target */
3   ompd_parallel_handle_t parallel_handle, /* IN: handle for a parallel region */
4   ompd_thread_handle_t *parent_thread_handle /* OUT: handle for parent thread */
5 );
```

Listing 1.6. Proposed signature for `ompd_get_anchestor_thread`

We propose to add the function `ompd_get_anchestor_thread` with a signature like in Listing 1.6 to the OMPD API. The signature is aligned to the API functions currently in the interface. We think, using the thread handle instead of the thread number is more consistent in case of the OMPD API.

7 Future Challenges

Although the OMPD API currently supports OpenMP 3.0 specification, when extended for the current 4.0 version of OpenMP, it will face a new set of challenges. In particular, the `target` construct whereby the application outsources its calculations to an accelerator will present technical challenges. In this section, we discuss how we prepare the current OMPD interface for the necessary future accelerators' support.

7.1 Context Pointer for Accelerators

In Section 5.4, we touched upon the topic of the meaning of a context pointer with respect to accelerators. We already discussed the need for a callback function to switch the context to the right location. When this callback is provided, an OMPD API function for an accelerator will first use this callback to switch the context from the accelerator thread context to the process thread context. As such switching would be necessary for each API call, it might be more efficient and cleaner to introduce a single API call thereby the OMPD library can specify the required context.

7.2 Addressing Accelerator Threads

Another potential issue is the specification of `ompd_osthread_handle`. The handle is important to build a mutual understanding on the low-level thread between the debugger and the OMPD library. The handle can be used when OMPD cannot determine whether or not a thread is an OpenMP thread by fetching a thread-local-storage (TLS) variable. On a runtime system that provides OpenMP thread personality through a TLS variable, we do not believe this handle is necessary. With accelerators, it is unlikely that all of the OpenMP runtimes will provide OpenMP personality via TLS. In many cases, the OpenMP thread running on an accelerator will be identified using the `osthread` handle. However, the current specification of the `osthread` handle will break compatibility if extended with accelerator support. Thus, we propose to use a flat struct that contains only an `int` to specify the kind of thread and an `uint64_t` for the specific thread handle. The values for the kinds of threads need to be defined in the interface.

7.3 Return Codes

All API and callback functions are specified to return an error code. The current specification provides one common set of error codes. If a callback function returns an error, and the API function fails, the debugger is interested in this error code. The set of error codes for callback functions should be a subset of error codes for API function calls.

8 Conclusions

In this paper, we described some of the issues that we experienced during implementing an OMPD library prototype. We proposed some changes to the OMPD technical report for both the callback and the API interface. The changes on the callback interface affect the ability of endianness conversion and type lookup. The proposed changes on the API interface concern the matching of compatible OpenMP runtime library and OMPD library versions and the specification of debugger breakpoints. We proposed to add a function to get the master thread in a parallel region. Finally, we highlighted certain aspects of the interface which will likely break compatibility between interface versions when extended for accelerator support in the future.

References

1. Allinea Software. Allinea DDT. `http://www.allinea.com/products/ddt`. Accessed 16 May 2015.
2. James Cownie, John DelSignore, Bronis R. de Supinski, and Karen Warren. Dmpl: An openmp dll debugging interface. In *Proceedings of the OpenMP Applications and Tools 2003 International Conference on OpenMP Shared Memory Parallel Programming*, WOMPAT'03. Springer-Verlag, 2003.
3. James Cownie and William Gropp. A Standard Interface for Debugger Access to Message Queue Information in MPI. In *Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 51–58, 1999.
4. Alexandre Eichenberger et. al. OMPT and OMPD: OpenMP Tools Application Programming Interfaces for Performance Analysis and Debugging. Technical report, OpenMP.org, May 2013. `http://openmp.org/mp-documents/ompt-tr.pdf`, Accessed 15 May 2015.
5. Alexandre Eichenberger et. al. OpenMP Technical Report 2 on the OMPT Interface. Technical report, OpenMP.org, March 2014. `http://openmp.org/mp-documents/ompt-tr2.pdf`, Accessed 15 May 2015.
6. Rogue Wave Software. TotalView® Graphical Debugger. `http://www.roguewave.com/products/totalview.aspx`, 2015. Accessed 16 May 2015.
7. Inc. Sun Microsystems. man pages section 3: Threads and realtime library functions. User documentation, May 2002. `https://docs.oracle.com/cd/E19683-01/816-0216/816-0216.pdf`, Accessed 15 May 2015.